CERT

Working with ROSE

David Svoboda

ROSE

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **OCT 2008**	2. REPORT TYPE	3. DATES COVERED **00-00-2008 to 00-00-2008**

4. TITLE AND SUBTITLE **Working with ROSE**	5a. CONTRACT NUMBER 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University ,Software Engineering Institute (SEI),Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S) 11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**58**	

Overview

The Problem:

- How to recognize Insecure Code?

Techniques:

- Automated Security Checking
- Static Analysis
- Abstract Syntax Tree (AST)
- ROSE

So how do we actually use ROSE?

Agenda

We will build a rule checker, showing how ROSE helps us.

- ROSE Setup
- ROSE Documentation
- Background
- Design
- Examining Source Code using ROSE
- Code
- Run & Test
- Useful ROSE Functions

What is ROSE?

Developed at Lawrence Livermore National Labs (LLNL)

- Analyzes program source code
- Produces Abstract Syntax Tree (AST)
- Can then be used for static analysis

We will use ROSE to enforce secure coding rules

`http://rosecompiler.org/`

Rosebud

Rosebud is a Virtual Machine that is useful for working with Rose.

- Rose and the checkers are already built; no need to compile
- Cross-platform, runs as a VM
- Includes popular developer tools (Eclipse, emacs, etc)

Rosebud 2

Download the ‘rosebud’ VM from

`rosecheckers.sourceforge.net`

You will need VMWare Player, to run Rosebud. VMWare Player is freely available at:

`downloads.vmware.com`

Extract the Rosebud package and start VM Player.

Rosebud 3

In VMPlayer, select *Open an Existing Virtual Machine*.

When it prompts you for a virtual machine (VM) to open, go to the **`rosebud`** **directory,** and

Select **`rosebud.vmx`**. This 'boots up' the Rosebud virtual machine. After a few seconds, a login prompt will appear.

Enter username: **`rose`** password: **`roserose`**

The system will then re-prompt you for the password, re-enter it.

The system will then give you a command-line prompt (a single **`%`)**

Type **`startx`** <*RETURN*>. This will bring up the GUI.

Rosebud 4

After desktop turns blue, right-click on the desktop. This brings up the program menu.

You should now be able to build and test the rules...you can do this with these commands in a terminal:

```
cd ~/src/rosecheckers
make tests
```

ROSE Setup on Andrew

Your environment should contain the following:

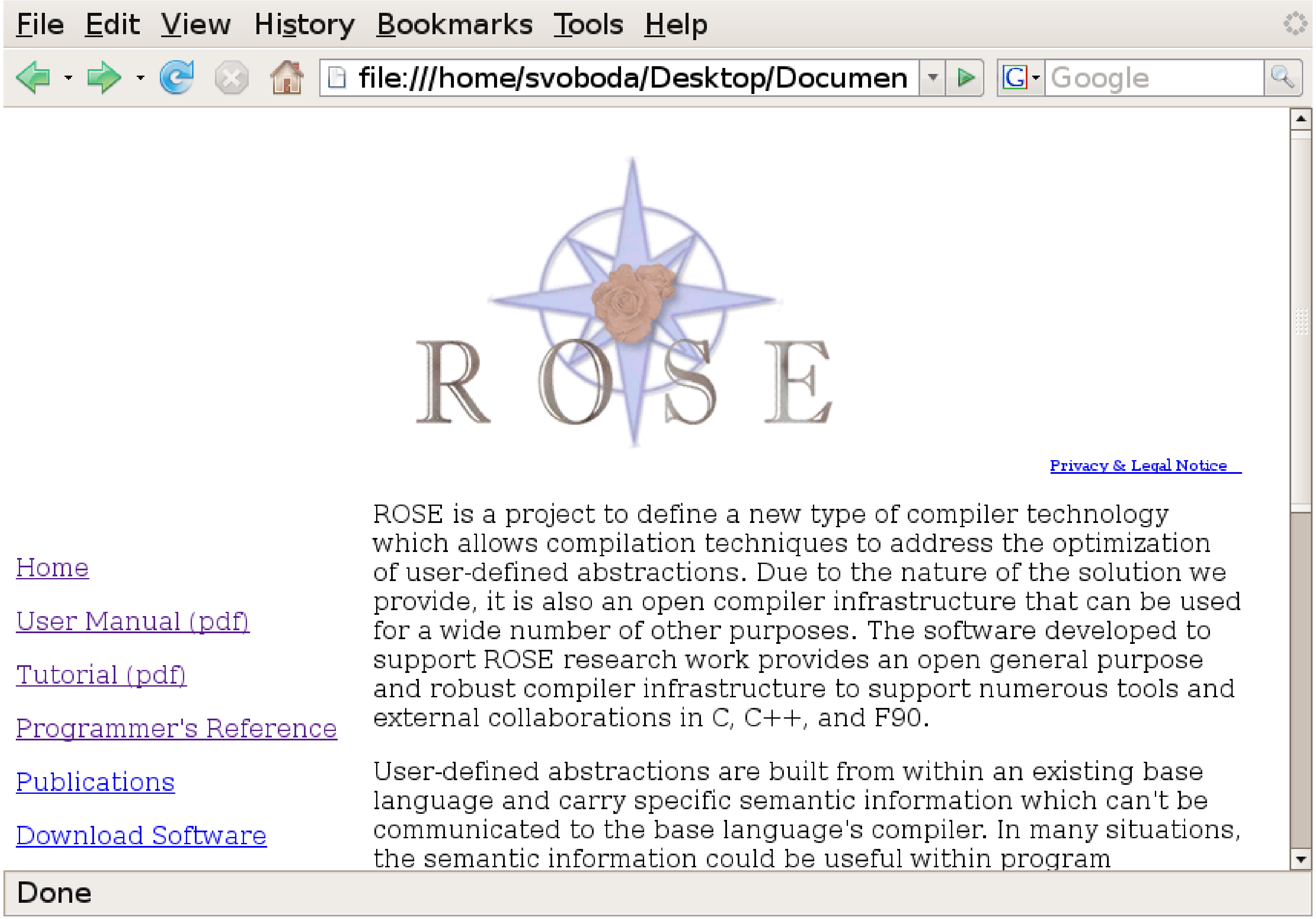
```
setenv ROSE /afs/andrew/usr/svoboda/public/rose
setenv LD_LIBRARY_PATH $ROSE/lib:$LD_LIBRARY_PATH
setenv PATH $ROSE/bin:$PATH
```

Check out the Rosecheckers project from SourceForge.

```
svn checkout
  https://anonymous@rosecheckers.svn.sourceforge.net/s
  vnroot/rosecheckers/trunk/rosecheckers
```

You should now be able to build and test the rules.

ROSE Homepage



ROSE is a project to define a new type of compiler technology which allows compilation techniques to address the optimization of user-defined abstractions. Due to the nature of the solution we provide, it is also an open compiler infrastructure that can be used for a wide number of other purposes. The software developed to support ROSE research work provides an open general purpose and robust compiler infrastructure to support numerous tools and external collaborations in C, C++, and F90.

User-defined abstractions are built from within an existing base language and carry specific semantic information which can't be communicated to the base language's compiler. In many situations, the semantic information could be useful within program

ROSE Documentation

User Manual

Full documentation for the Rose features and techniques

Tutorial

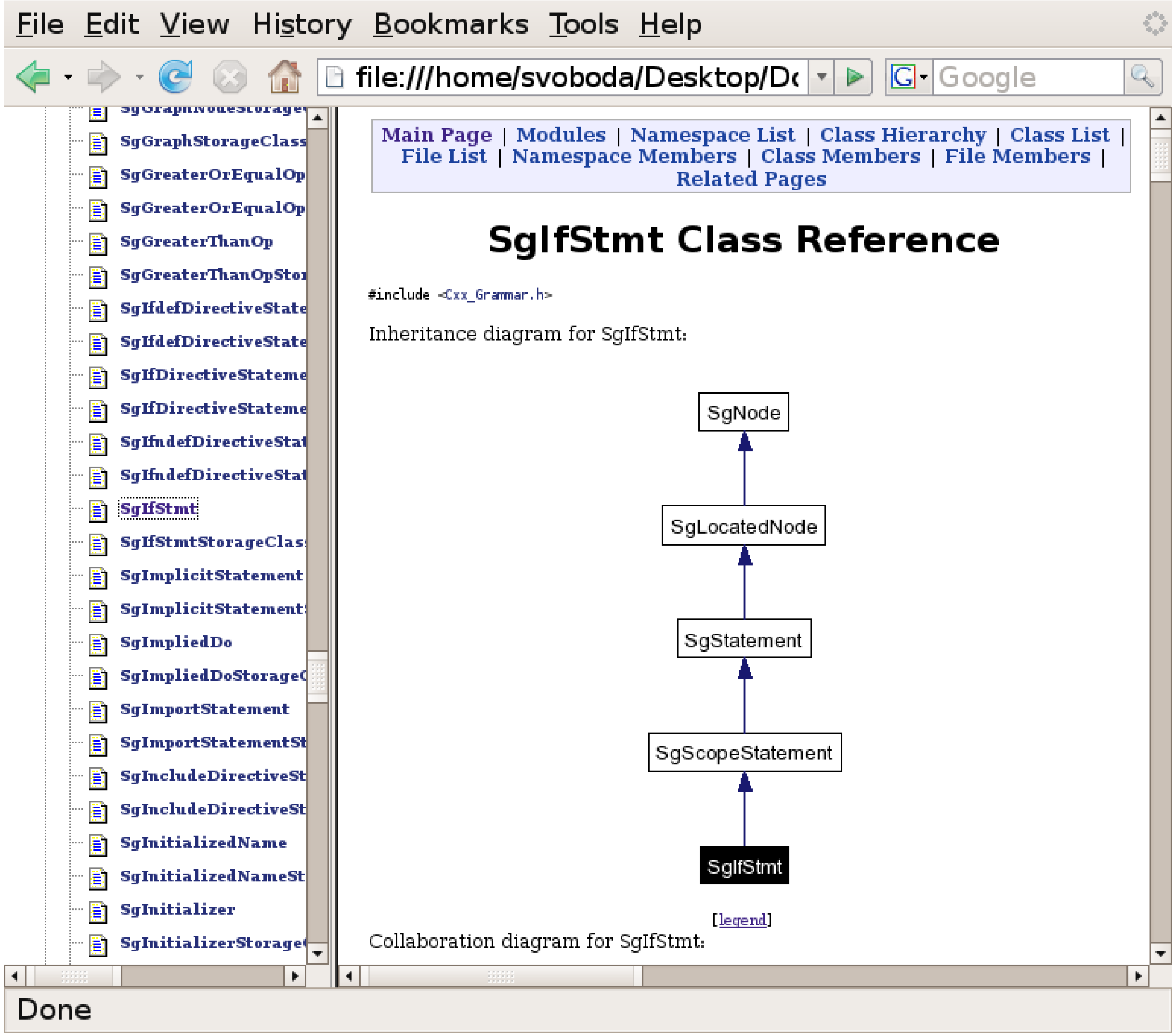
Guide to installing ROSE and some of its utilities

Programmer’s Reference

Web-based documentation for each class and method in ROSE.

Generated by Doxygen

Programmer’s Reference 1


File Edit View History Bookmarks Tools Help
file:///home/svoboda/Desktop/Dc
Google
Done

- SgGraphNodeStorage
- SgGraphStorageClass
- SgGreaterOrEqualOp
- SgGreaterOrEqualOp
- SgGreaterThanOp
- SgGreaterThanOpStor
- SgIfdefDirectiveState
- SgIfdefDirectiveState
- SgIfDirectiveStateme
- SgIfDirectiveStateme
- SgIfndefDirectiveStat
- SgIfndefDirectiveStat
- SgIfStmt
- SgIfStmtStorageClas
- SgImplicitStatement
- SgImplicitStatement
- SgImpliedDo
- SgImpliedDoStorage
- SgImportStatement
- SgImportStatementSt
- SgIncludeDirectiveSt
- SgIncludeDirectiveSt
- SgInitializedName
- SgInitializedNameSt
- SgInitializer
- SgInitializerStorage

Main Page | Modules | Namespace List | Class Hierarchy | Class List | File List | Namespace Members | Class Members | File Members | Related Pages

SgIfStmt Class Reference

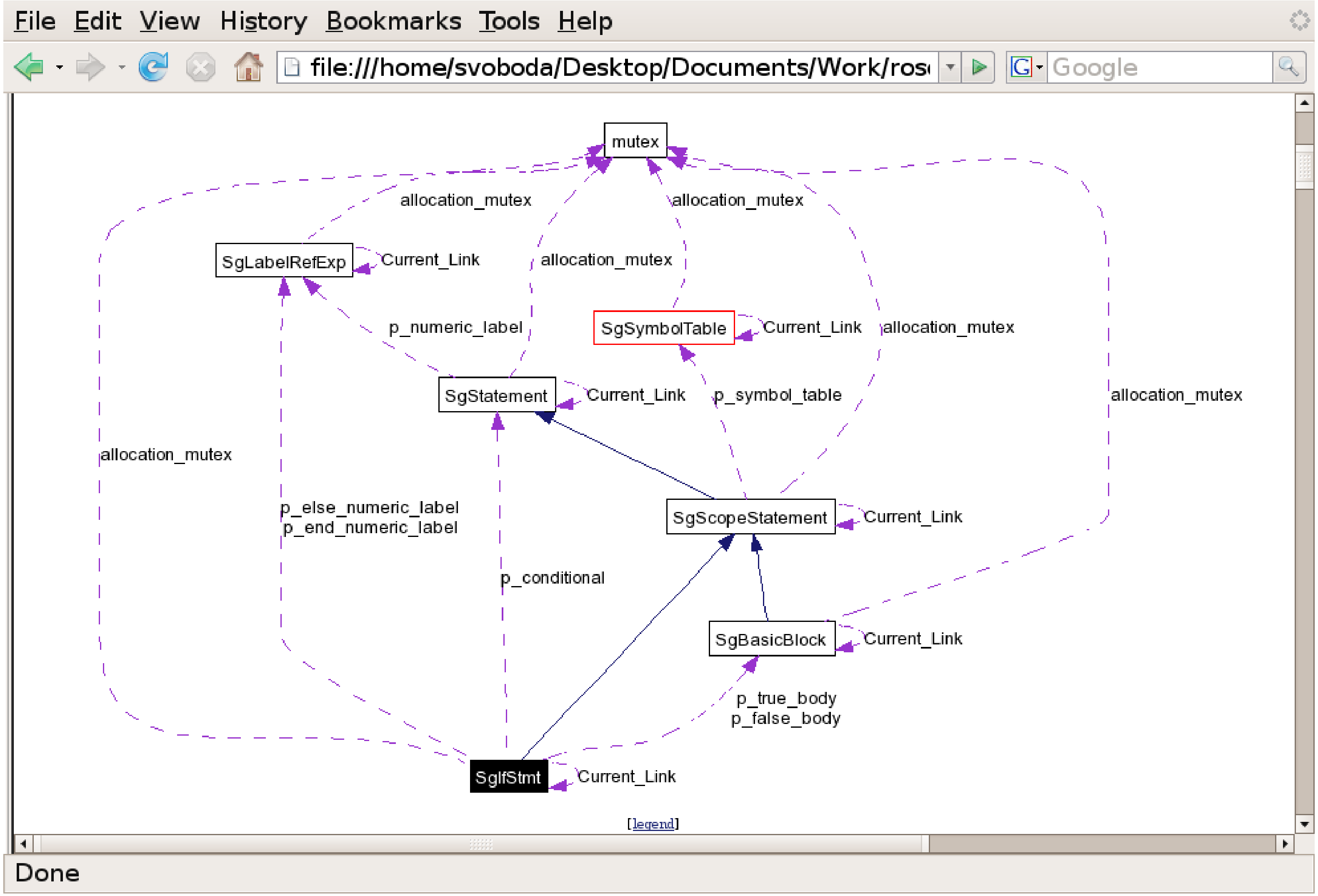
```
#include <Cxx_Grammar.h>
```

Inheritance diagram for SgIfStmt:

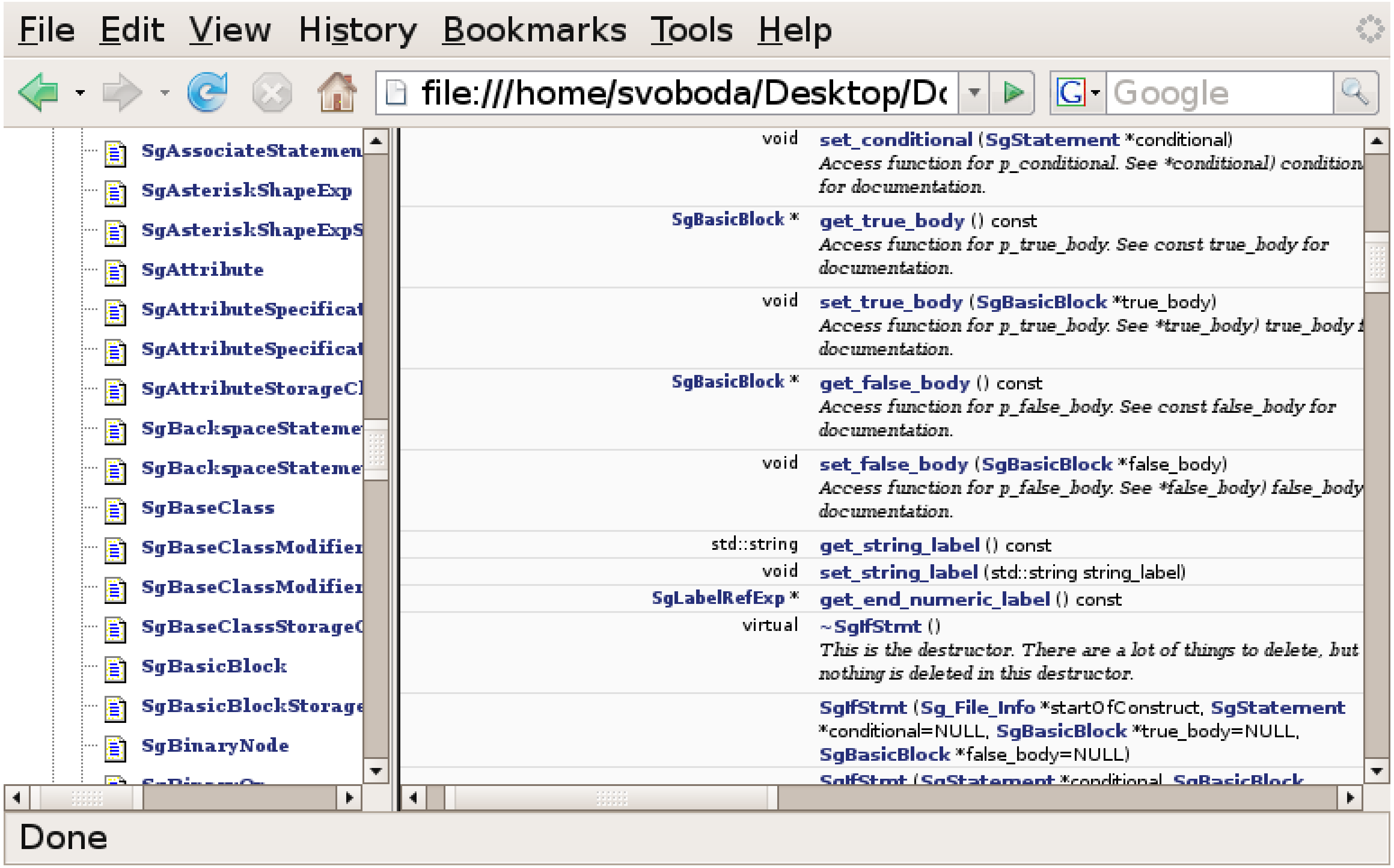
[legend]

Collaboration diagram for SgIfStmt:

Programmer’s Reference 2



Programmer's Reference 3

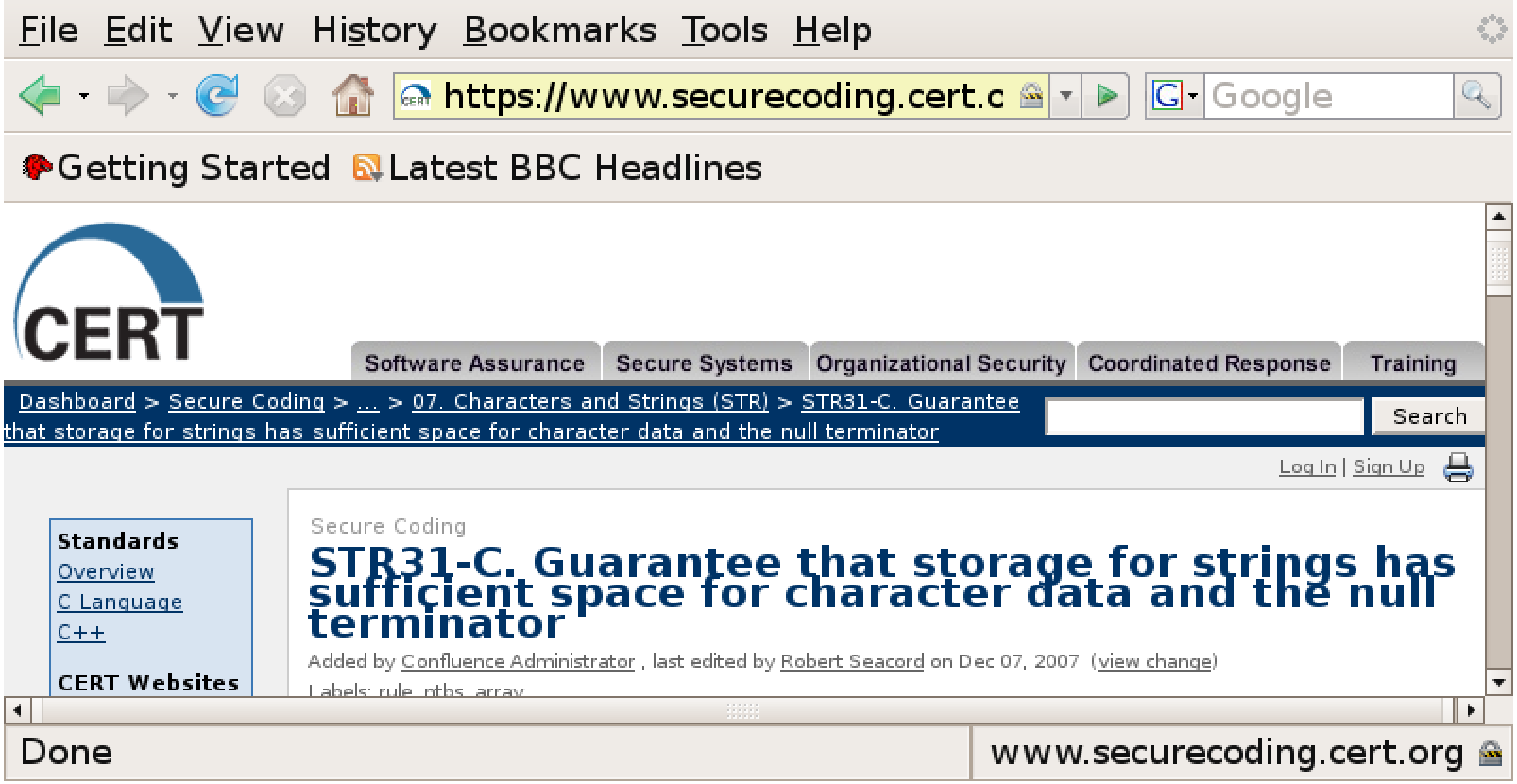

File Edit View History Bookmarks Tools Help
file:///home/svoboda/Desktop/Dc
Google
Done

SgAssociateStatemen
SgAsteriskShapeExp
SgAsteriskShapeExpS
SgAttribute
SgAttributeSpecificat
SgAttributeSpecificat
SgAttributeStorageCl
SgBackspaceStateme
SgBackspaceStateme
SgBaseClass
SgBaseClassModifier
SgBaseClassModifier
SgBaseClassStorageC
SgBasicBlock
SgBasicBlockStorage
SgBinaryNode

void	**set_conditional** (**SgStatement** *conditional) *Access function for p_conditional. See *conditional) condition for documentation.*
SgBasicBlock *	**get_true_body** () const *Access function for p_true_body. See const true_body for documentation.*
void	**set_true_body** (**SgBasicBlock** *true_body) *Access function for p_true_body. See *true_body) true_body f documentation.*
SgBasicBlock *	**get_false_body** () const *Access function for p_false_body. See const false_body for documentation.*
void	**set_false_body** (**SgBasicBlock** *false_body) *Access function for p_false_body. See *false_body) false_body documentation.*
std::string	**get_string_label** () const
void	**set_string_label** (std::string string_label)
SgLabelRefExp *	**get_end_numeric_label** () const
virtual	**~SgIfStmt** () *This is the destructor. There are a lot of things to delete, but nothing is deleted in this destructor.*
	SgIfStmt (**Sg_File_Info** *startOfConstruct, **SgStatement** *conditional=NULL, **SgBasicBlock** *true_body=NULL, **SgBasicBlock** *false_body=NULL)

Building a Rule Checker

We'll study rule STR31-C



Dashboard > Secure Coding > ... > 07. Characters and Strings (STR) > STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator

Secure Coding

STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator

Added by Confluence Administrator , last edited by Robert Seacord on Dec 07, 2007 (view change)

Labels: rule, ntbs, array

Test Cases

Before coding, we need at least one positive test case and one negative test case, These will prove to us that the code works.

The **`test`** directory contains **c**ompliant and noncompliant code **e**xamples…all compliant examples pass all the secure coding rules. The noncompliant code examples each fail a single secure coding rule.

Our first two test files will be **`test/c.ncce.wiki.STR.c`** and **`test/c.cce.wiki.STR.c`**

Non-Compliant Code Example

```
#include <string.h>
#include <stdlib.h>

int main() {
  /* ... */
  char buff[256];
  strcpy(buff, getenv("EDITOR"));
  /* ... */
  return 0;
}
```

From `test/c.ncce.wiki.STR.c`

Compliant Code Example

```
#include <string.h>
#include <stdlib.h>

int main() {
  /* ... */
  char* editor;
  char* buff;
  editor = getenv(“EDITOR”);
  if (editor) {
    buff = (char*) malloc(strlen(editor)+1);
    if (!buff) {
      /* handle malloc() error */
    }
    strcpy( buff, editor);
  }
  /* ... */
  return 0;
}
```

From `test/c.cce.wiki.STR.c`

Design Idea

```
#include <string.h>
#include <stdlib.h>

int main() {
  /* ... */
  char buff[256];
  strcpy(buff, getenv("EDITOR"));
  /* ... */
  return 0;
}
```

An attacker can compromise the system by setting the EDITOR environment variable to a string larger than 256 chars!

2nd arg to `strcpy()` is a `char*`

`getenv()` makes no promise about the size of the string it returns!

1st arg to `strcpy()` is a local `char[]`

We could flag any instance of ***`strcpy()`*** *where the 1st arg is a local fixed array and the 2nd arg is a pointer.*

Other Test Cases

STR31-C has many other positive and negative examples, which we could include when testing our rule.

- Can we test them all?
- Will our idea of checking **`strcpy()`**’s arguments work on them?
- If not, how can we check them

Non-Compliant Code Example: (off-by-1)

```
char dest[ARRAY_SIZE];
char src[ARRAY_SIZE];
size_t i;
/* ... */
for (i=0; src[i] &&
    (i < sizeof(dest)); i++) {
  dest[i] = src[i];
}
dest[i] = '\0'; /* ... */
```

Non-Compliant Code Example: (strcpy())

```
int main(int argc, char *argv[]) {
  /* ... */
  char prog_name[128];
  strcpy(prog_name, argv[0]);
  /* ... */
}
```

Compliant Code Example: (strcpy_s())

```
int main(int argc, char *argv[]) {
  /* ... */
  char * prog_name;
  size_t prog_size;
  prog_size = strlen(argv[0])+1;
  prog_name = (char *)malloc(prog_size);
  if (prog_name != NULL) {
    if (strcpy_s(prog_name, prog_size, argv[0])) {
      /* Handle strcpy_s() error */
    }
  } else {
    /* Couldn't get the memory - recover */
  }
  /* ... */
}
```

Testing Conclusions

- We can’t handle the off-by-1 example with our design at all.
- Our current design will work on the strcpy() example without any modifications.
  - We should add the strcpy() example to our test suite, in `test/c.ncce.wiki.STR.c`
- Our design won’t work on the strcpy_s() example, but we could always extend it to recognize the arguments to strcpy_s() as well as strcpy().
  - We should note this as a task to be done later.

Checker Design for STR31-C

1. Traverse AST.
2. For each `strcpy()` function call
   1. Get both arguments to `strcpy()`. If
   2. 1st argument is a variable AND
   3. the variable's type is a fixed-length array AND
   4. 2nd argument's type is NOT a fixed-length array
      - **Report a violation of STR31-C!**

Design Limitations

- Will report all cases of **`strcpy( char[], char*)`**, including false positives.
- Will not report any other cases of **`strcpy()`**, including false negatives
- Will not catch other string-copy functions like **`strncpy()`**, **`strcpy_s()`**, or **`memcpy()`**.
- Will not catch string-copying done ‘by hand’ (for instance, our off-by-1 example)

Design Conclusions

- Designing checkers helps to ‘flesh out’ secure coding rules.
- Be aware of
  - false positives
  - false negatives
- A checker does not need to be complete to be useful.
- It’s OK to write more than one checker for a rule.
- Don’t worry about pathological cases, focus primarily on violations likely to occur ‘in the wild’.

ROSE in Action

When we run our ROSE program, called **`diagnose`**, on our insecure source code, we get an error message:

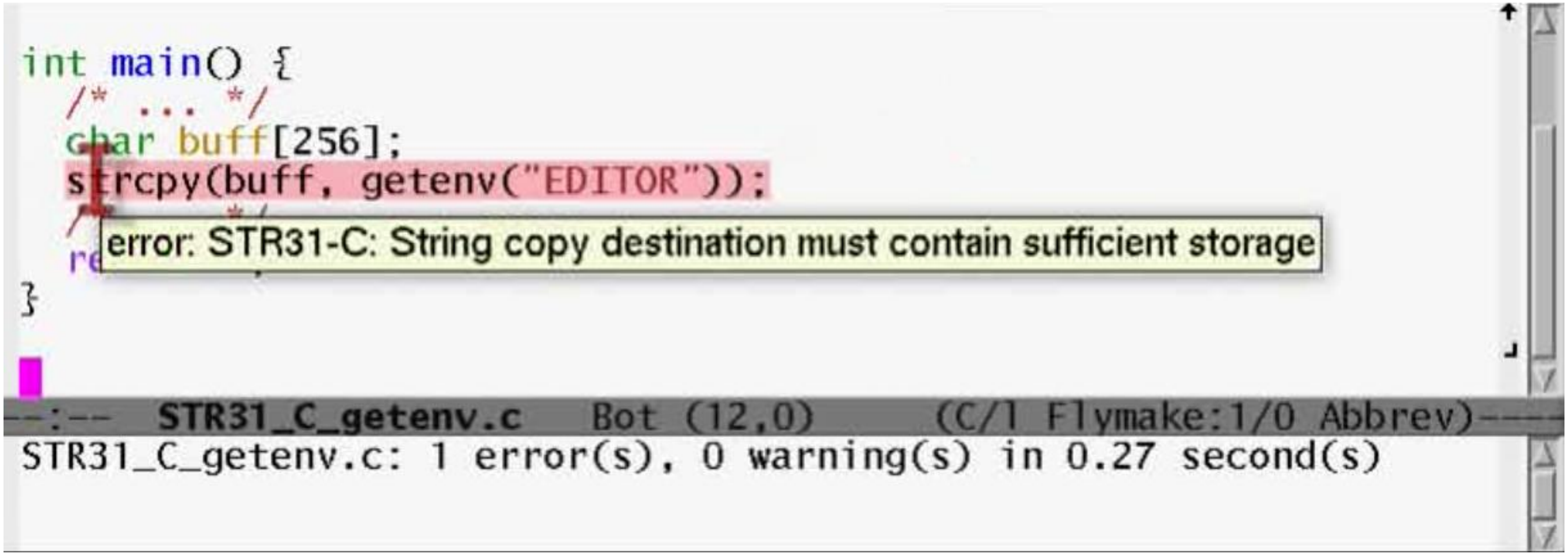
```
% ./rosecheckers test/c.ncce.wiki.STR.c
c.ncce.wiki.STR.c:7: error: STR31-C: String copy
destination must contain sufficient storage
%
```

If we run **`rosecheckers`** on a secure program, we get no output:

```
% ./rosecheckers test/c.cce.wiki.STR.c
%
```

So our **`rosecheckers`** program acts like a compiler, or **`lint`**.

ROSE integrated with Emacs



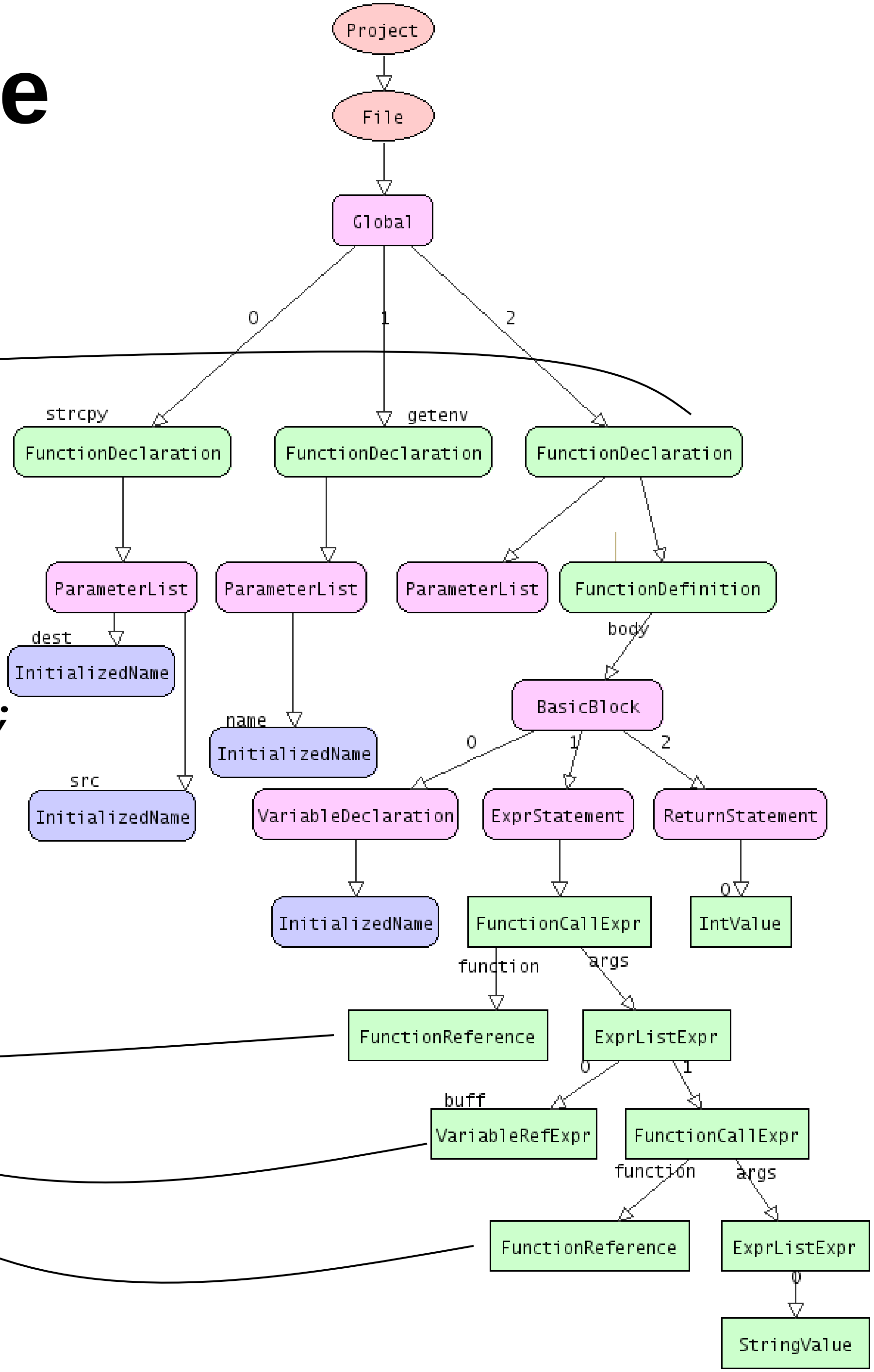
```
int main() {
  /* ... */
  char buff[256];
  strcpy(buff, getenv("EDITOR"));
  /
  re
}
```

error: STR31-C: String copy destination must contain sufficient storage

```
-:--   STR31_C_getenv.c   Bot (12,0)      (C/l Flymake:1/0 Abbrev)----
STR31_C_getenv.c: 1 error(s), 0 warning(s) in 0.27 second(s)
```

Example Source Code

```
#include <string.h>
#include <stdlib.h>

int main() {
  /* ... */
  char buff[256];
  strcpy(buff, getenv("EDITOR"));
  /* ... */
  return 0;
}
```



Source Code Syntax Tree

The command

```
	cpp2ps -t foo.c foo.c.ps
	dot2ps foo.c.dot
```

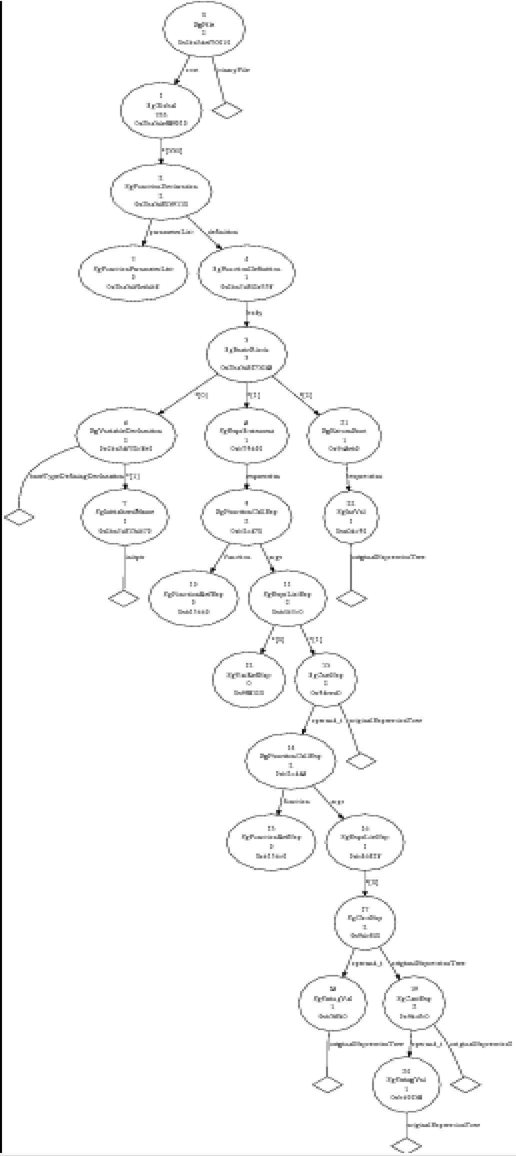
produce a PostScript file **`foo.c.dot.ps`** that contains the Abstract Syntax Tree (AST) of the source code.

On rosebud, the **`gv`** program can be used to view PostScript files.

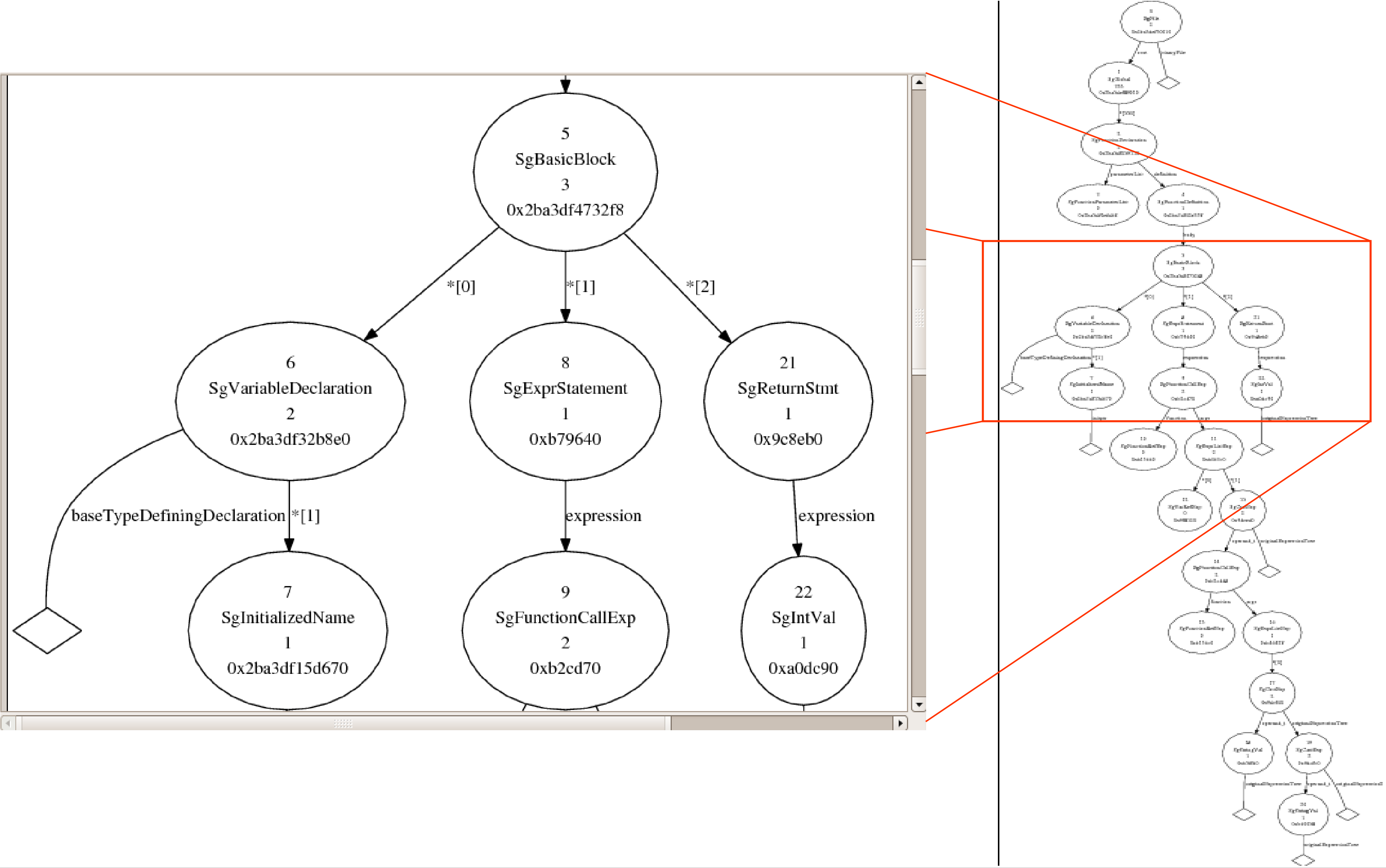
```
	gv foo.c.dot.ps
```

Abstract Syntax Tree 1

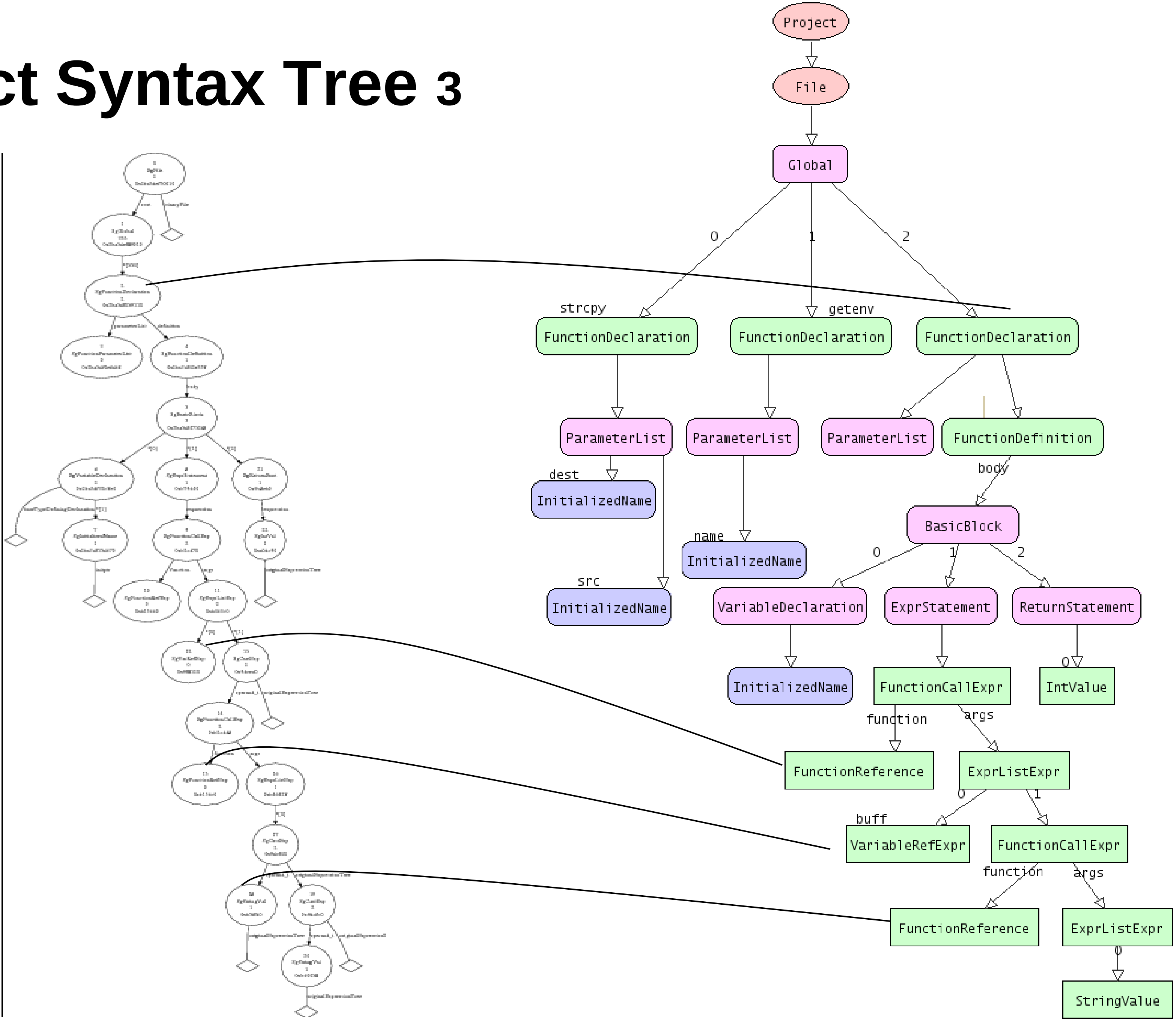
```
#include <string.h>
#include <stdlib.h>

int main() {
  /* ... */
  char buff[256];
  strcpy(buff, getenv("EDITOR"));
  /* ... */
  return 0;
}
```



Abstract Syntax Tree 2



Abstract Syntax Tree 3



AST Attributes

The command

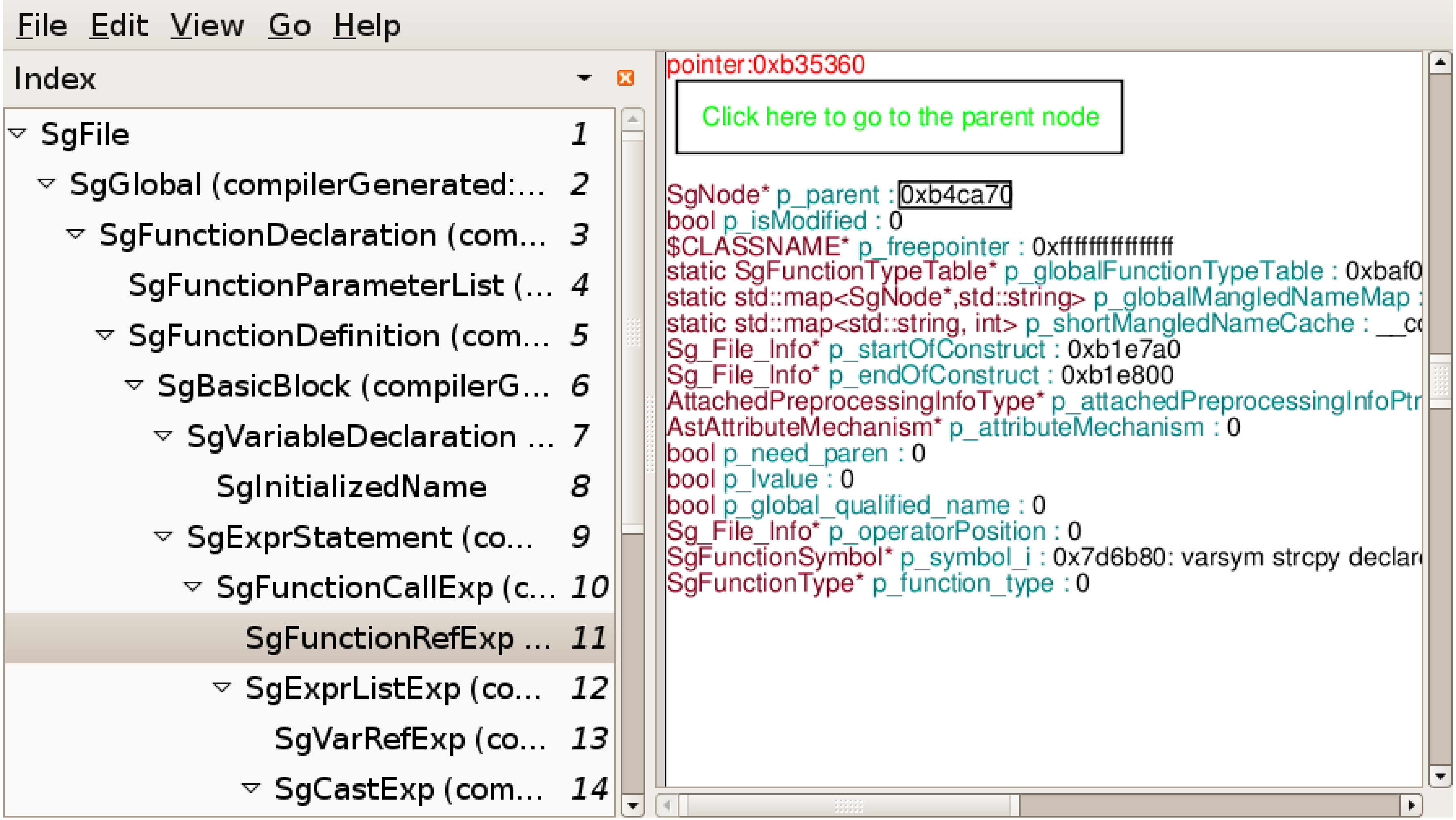
```
cpp2pdf foo.c
```

produces a PDF **`foo.c.pdf`** that contains the source code AST, and also shows each class's attributes

On rosebud, the **`xpdf`** program can be used to view PDFs.

```
xpdf foo.c.pdf
```

AST Attributes cont.



```
SgFile 1
  SgGlobal (compilerGenerated:... 2
    SgFunctionDeclaration (com... 3
      SgFunctionParameterList (... 4
      SgFunctionDefinition (com... 5
        SgBasicBlock (compilerG... 6
          SgVariableDeclaration ... 7
            SgInitializedName 8
          SgExprStatement (co... 9
            SgFunctionCallExp (c... 10
              SgFunctionRefExp ... 11
              SgExprListExp (co... 12
                SgVarRefExp (co... 13
                SgCastExp (com... 14
```

```
pointer:0xb35360
Click here to go to the parent node

SgNode* p_parent : 0xb4ca70
bool p_isModified : 0
$CLASSNAME* p_freepointer : 0xffffffffffffffff
static SgFunctionTypeTable* p_globalFunctionTypeTable : 0xbaf0
static std::map<SgNode*,std::string> p_globalMangledNameMap :
static std::map<std::string, int> p_shortMangledNameCache : __cc
Sg_File_Info* p_startOfConstruct : 0xb1e7a0
Sg_File_Info* p_endOfConstruct : 0xb1e800
AttachedPreprocessingInfoType* p_attachedPreprocessingInfoPtr
AstAttributeMechanism* p_attributeMechanism : 0
bool p_need_paren : 0
bool p_lvalue : 0
bool p_global_qualified_name : 0
Sg_File_Info* p_operatorPosition : 0
SgFunctionSymbol* p_symbol_i : 0x7d6b80: varsym strcpy declar
SgFunctionType* p_function_type : 0
```

Whole Syntax Tree

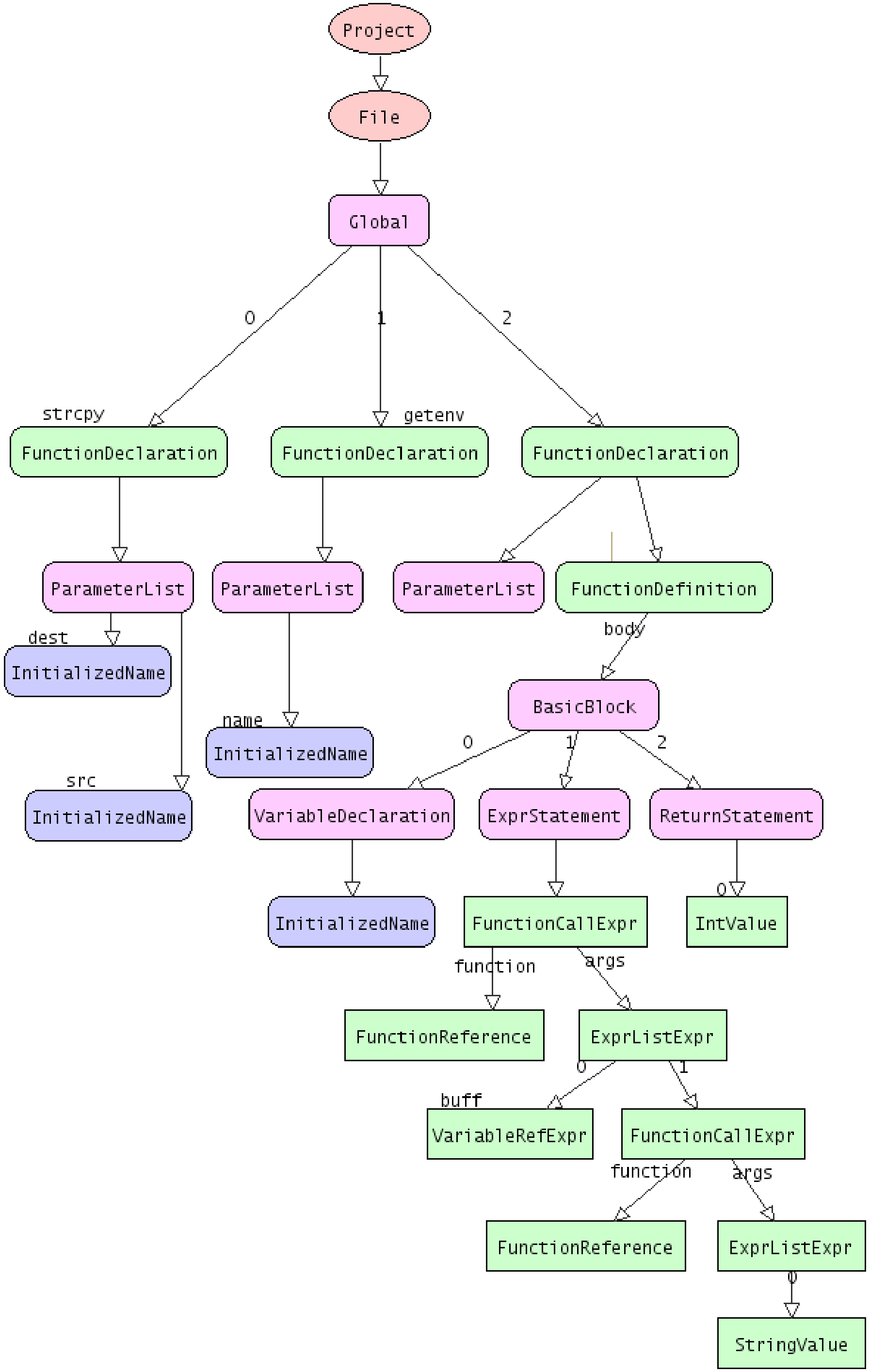
The AST does not contain semantic information, such as:

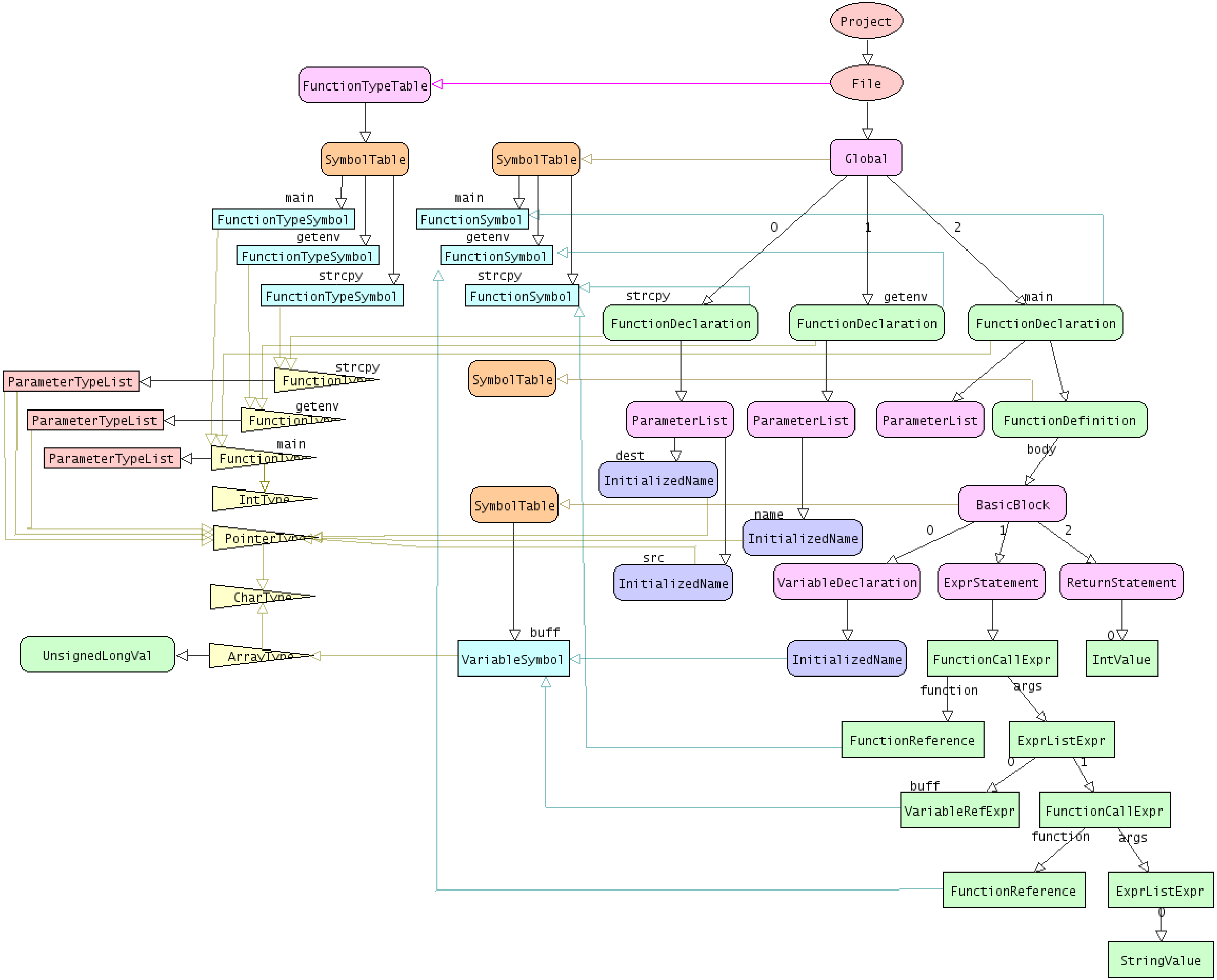
- Type Definitions
- Symbol Tables
- Variable Definitions

The Whole AST adds these bits of information to the AST.

Whole Syntax Tree 2

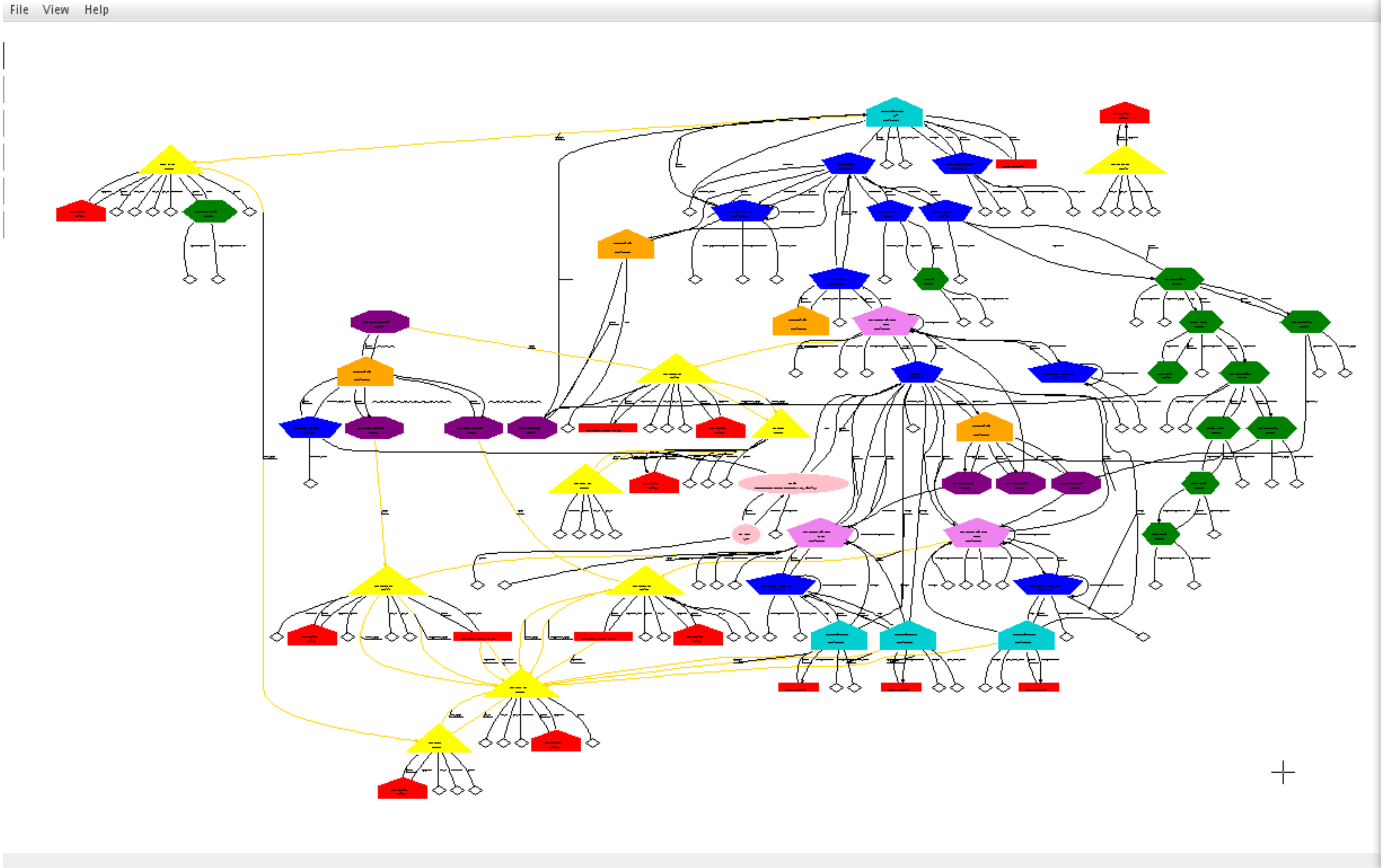
```
char* strcpy(char*, char*);
char* getenv(char*);

int main() {
  char buff[256];
  strcpy(buff, getenv("EDITOR"));
  return 0;
}
```



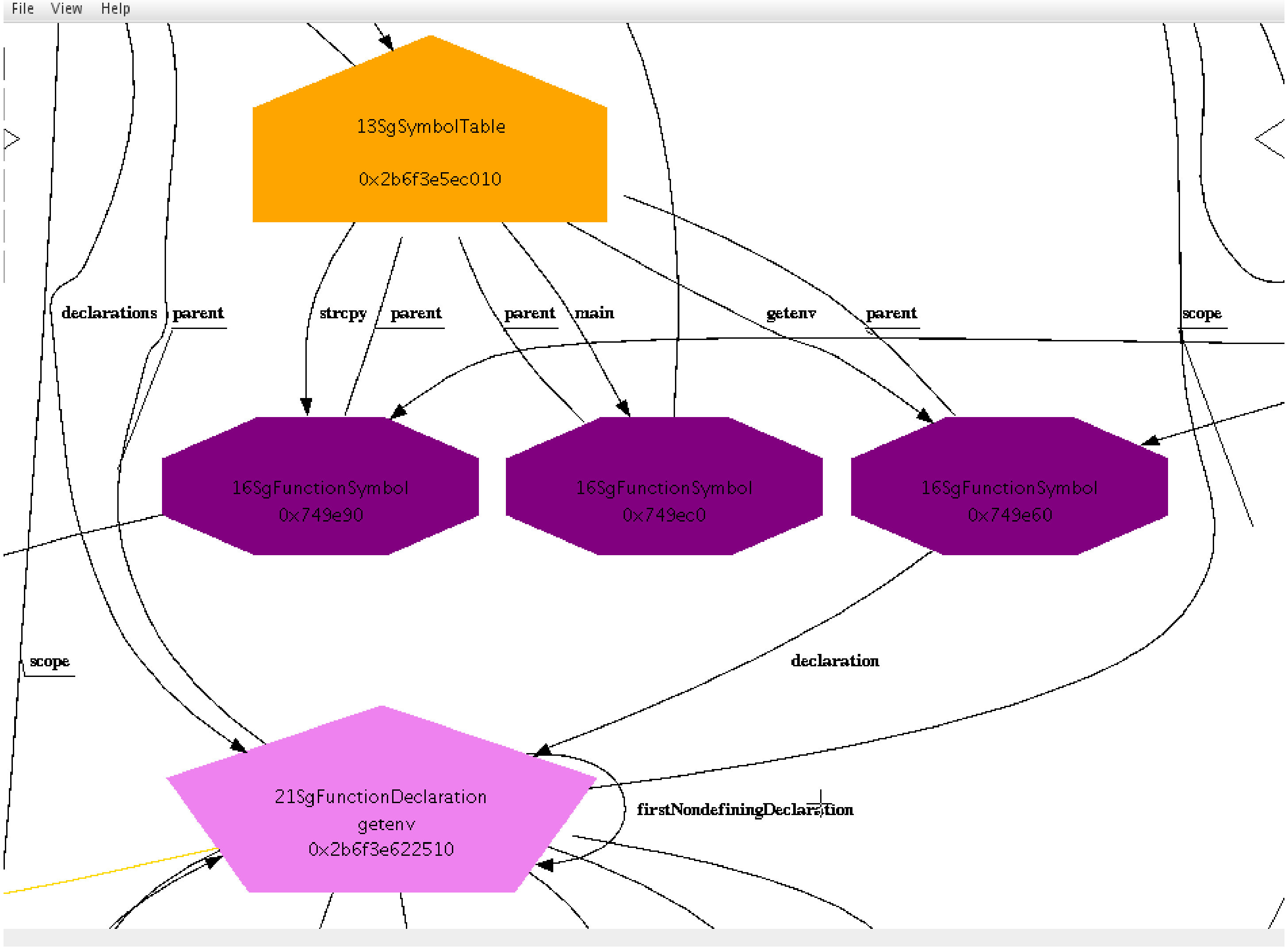


Project
File
FunctionTypeTable
Global
SymbolTable
SymbolTable
main
FunctionTypeSymbol
getenv
FunctionTypeSymbol
strcpy
FunctionTypeSymbol
main
FunctionSymbol
getenv
FunctionSymbol
strcpy
FunctionSymbol
0
1
2
strcpy
getenv
main
FunctionDeclaration
FunctionDeclaration
FunctionDeclaration
strcpy
ParameterTypeList
FunctionType
getenv
ParameterTypeList
FunctionType
main
ParameterTypeList
FunctionType
IntType
PointerType
CharType
UnsignedLongVal
ArrayType
SymbolTable
SymbolTable
ParameterList
ParameterList
ParameterList
FunctionDefinition
dest
InitializedName
body
BasicBlock
name
InitializedName
src
InitializedName
0
1
2
VariableDeclaration
ExprStatement
ReturnStatement
buff
VariableSymbol
InitializedName
FunctionCallExpr
0
IntValue
function
args
FunctionReference
ExprListExpr
0
1
buff
VariableRefExpr
FunctionCallExpr
function
args
FunctionReference
ExprListExpr
0
StringValue

Whole Syntax Tree 3



Whole Syntax Tree 4



How `rosecheckers` Uses ROSE

```
#include “rose.h”
#include “utilities.h”

int main( int argc, char* argv[]) {
  SgProject* project = frontend(argc,argv);
  ROSE_ASSERT( project );
  visitorTraversal exampleTraversal;
  exampleTraversal.traverseInputFiles(
               project, preorder);
  return 0;
}
```

ROSE parses source code

Traverse AST, examine each node.

AST Node Analysis

This is called for each node in the AST:

```
bool EXP(const SgNode *node) {
  bool violation = false;
  violation |= EXP01_A(node);
  violation |= EXP09_A(node);
  violation |= EXP34_C(node);
  return violation;
}
```

Each routine here enforces a single CERT Secure Coding Rule, and returns true if the node indicates a violation.

Similar code exists for other sections **STR**, **MEM**, etc

ROSE Checker Skeleton

```
#include "rose.h"
#include "utilities.h“

bool STR31_C(const SgNode *node ) {
   // ensure sufficient storage for strings
      /* ??? */
}

bool STR(const SgNode *node) {
  bool violation = false;
  /* … */
  violation |= STR31_C(node);
  return violation;
}
```

This routine will be called for every node in the AST. We want it to print an error message and return **`true`** exactly once when run on our non-compliant example.

Current Status

How do we do this?

```
bool STR31_C(const SgNode *node )
{
  /* ??? */
}

bool STR(const SgNode *node) {
  bool violation = false;
  /* … */
  violation |= STR31_C(node);
  return violation;
}
```

Traverse AST.

DONE

For each `strcpy()` function call

1. Get both arguments to `strcpy()`. If
2. 1st argument is a variable AND
3. the variable's type is a fixed-length array AND
4. 2nd argument's type is NOT a fixed-length array
   - **Report a violation of STR31-C!**

Utility Functions from `utilities.h`

```
// Returns non-NULL if node is a call of
// function with given name
const SgFunctionSymbol *isCallOfFunctionNamed(
       const SgNode *node, const std::string &name);

// Returns reference to ith argument
// of function reference. Dives through typecasts.
// Returns NULL if no such parm
const SgExpression* getFnArg(
      const SgFunctionRefExp* node, int i);

void print_error(
      const SgNode* node, const char* rule,
      const char* desc,bool warning = false);
```

Current Status

```
bool STR31_C(const SgNode *node )
{
 if (!isCallOfFunctionNamed(
node, "strcpy"))
    return false;

   /* ??? */
}
```

At this point, node will always point to a `strcpy()` function call

Traverse AST.

For each `strcpy()` function call

DONE

1. Get both arguments to `strcpy()`. If
2. 1st argument is a variable AND
3. the variable's type is a fixed-length array AND
4. 2nd argument's type is NOT a fixed-length array
   - **Report a violation of STR31-C!**

The `isSg` Family

A set of useful functions that are useful for typecasting a **`SgNode*`** into an appropriate node type. They return **`NULL`** if the node is the wrong type.

```
const SgNode* node;
const SgFunctionRefExp* sig_fn
  = isSgFunctionRefExp( node);
if (sig_fn == NULL) {
  cerr << "Node is not a “
       << “SgFunctionRefExp!" << endl;
}
```

Current Status

```
bool STR31_C(const SgNode *node )
{
 if (!isCallOfFunctionNamed(
node, "strcpy"))
    return false;

 const SgVarRefExp* ref =
    isSgVarRefExp( getFnArg(
isSgFunctionRefExp(node), 0));
  if (ref == NULL)
    return false;

  /* ??? */
}
```

At this point, `ref` refers to the 1st arg of `strcpy()` and it is a variable.

Traverse AST.

For each `strcpy()` function call

1. Get both arguments to `strcpy()`. If
2. 1st argument is a variable AND
3. the variable’s type is a fixed-length array AND
4. 2nd argument’s type is NOT a fixed-length array

DONE

- **Report a violation of STR31-C!**

Current Status

```
bool STR31_C(const SgNode
*node ) {
 if (!isCallOfFunctionNamed(
node, "strcpy"))
    return false;

 const SgVarRefExp* ref =
    isSgVarRefExp( getFnArg(
isSgFunctionRefExp(node), 0));
  if (ref == NULL)
    return false;
 if (!Type( getRefDecl(
>get_type()).isArray())
    return false;

  /* ??? */
}
```

Traverse AST.

For each **`strcpy()`** function call

1. Get both arguments to **`strcpy()`**. If
2. 1st argument is a variable AND
3. the variable's type is a fixed-length array AND
4. 2nd argument's type is NOT a fixed-length array

DONE

- **Report a violation of STR31-C!**

Current Status

```
const SgVarRefExp* ref =
    isSgVarRefExp(
getFnArg(
isSgFunctionRefExp(node),
0));
  if (ref == NULL)
    return false;
  if (!Type( getRefDecl(
ref)-
>get_type()).isArray())
    return false;
  if (Type( getFnArg(
isSgFunctionRefExp(node),
1)->get_type()).isArray())
    return false;
```

DONE

Traverse AST.

For each **strcpy()** function call

1. Get both arguments to **strcpy()**. If
2. 1st argument is a variable AND
3. the variable’s type is a fixed-length array AND
4. 2nd argument’s type is NOT a fixed-length array
   - **Report a violation of STR31-C!**

ROSE Checker for STR31-C

```
#include "rose.h"
#include "utilities.h‟

bool STR31_C(const SgNode *node ) {
  // ensure sufficient storage for strings
  if (!isCallOfFunctionNamed( node, "strcpy"))
    return false;

  const SgVarRefExp* ref =
    isSgVarRefExp( getFnArg( isSgFunctionRefExp(node), 0));
  if (ref == NULL)
    return false; // strcpy() not copying into simple var
  if (!Type( getRefDecl( ref)->get_type()).isArray())
    return false;
  if (Type( getFnArg( isSgFunctionRefExp(node),
                      1)->get_type()).isArray())
    return false;

  print_error( node, "STR31-C", "String copy destination must contain
sufficient storage");
  return true;
}
```

Called for every node in the AST.

We have an instance of ***strcpy()***

1st arg is a local fixed array

2nd arg is a pointer (eg NOT an array

Build and Test

To rebuild **`rosecheckers`** with a new rule, type

```
make pgms
```

To test **`rosecheckers`** on all rules, type

```
make tests
```

Testing New Rule

When run on the bad example, rosecheckers produces an error message:

```
% ./rosecheckers test/c.ncce.wiki.EXP.c
EXP.c:5: error: EXP09-A: malloc called using something other than sizeof()
%
```

When run on the good example, rosecheckers produces nothing.

```
% ./rosecheckers test/c.cce.wiki.EXP.c
%
```

Useful ROSE Functions

- `isSg***(node)`
- `unparseToString()`
- `querySubTree(SgNode* node, type)`

unparseToString()

Returns a string representation of the source code associated with a node. Useful for debugging:

```
const SgNode* node;
cout << "Node: “
     << node->unparseToString()
     << endl;
```

querySubTree(node, type)

Traverses the AST that descends from **`node`**. Returns a list (a **`std::vector`**, actually) of all subnodes of appropriate type.

```
const SgNode* node;
Rose_STL_Container<SgNode *> nodes
  = NodeQuery::querySubTree(
        const_cast<SgNode*>( def), V_SgVarRefExp);
Rose_STL_Container<SgNode*>::iterator i;
for (i = nodes.begin(); i != nodes.end(); ++i) {
  cout << "A SgVarRefExp: “
         << (*i)->unparseToString() << endl;
}
```

Note that **`querySubTree`** requires a non-**`const SgNode*`** as 1st argument.

Questions